1981

A General Theory of Automatic Program Synthesis

Carl H. Smith

Report Number:
80-360

A General Theory of Automatic Program Synthesis

Carl H. Smith [1]

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907
CSD TR 360
March 1981

ABSTRACT

Some results concerning inductive inference are surveyed. These results are interpreted with respect to automatic program synthesis, a special case of algorithmic inductive inference. The interpretations reinforce and refine opinions concerning automatic program synthesis, and artificial intelligence in general, which have been previously expressed in [9] and [15]. The final section digresses from science to speculation as the interpretations of the theorems surveyed are extended to differentiate various cognitive styles of learning.

1. Supported by NSF grant MCS-7903912

A General Theory of Automatic Program Synthesis

Carl H. Smith [1]

Department of Computer Sciences

Purdue University

West Lafayette, Indiana 47907

CSD TR 360

March 1981

1. Introduction

Herein a general theory of automatic program synthesis is presented. The results presented have all appeared elsewhere and are due to the author and others. The novelty of this paper is its presentation of theoretical results, some of which are technically intricate and/or obscure, to practitioners who implement algorithms which synthesize programs. These results delimit the ultimate capabilities of automatic program synthesizers. The relevance of this work to researchers in artificial intelligence is communicated below firstly by way of an analogy with early results in abstract computability theory and also by relating the theorems presented to extant AI work. Valuable intuitions concerning the nature of computation in general were formulated during the 60's (see the preface to either [12] or [11]). Perhaps the best known of these results is the speed-up theorem in [5]. Occasionally, abstract theoretical results are incorporated directly into the practice of computing. For example, the first batch operating systems had, as a consequence of Turing's results [16], a facility to apriori bound the amount of time taken to run any given job. The refinement of results from abstract complexity theory has led to the development of several

algorithms which are widely used today (see the chapter by Borodin in [1]). Just as the early results in computability theory gave practitioners valuable insights into the systems they were designing, the preliminary results concerning the general theory of automatic program synthesis may be enlightening to those currently implementing program synthesizers.

Of particular interest is the similarity between the conclusions we draw from rigorous theorems and the conclusions of some researchers doing more practical AI research. For example, Sussman [15] writes

> "Many people working on automatic program synthesis expect that a system can be built which will, given a description of the problem to be solved, synthesize a "correct" program. I believe that this approach is a mistake. In real situations the complete specification of a problem is unknown, and what we really see happening is an evolutionary process."

Not only are the theorems stated below consonant with Sussman's views, they also rigorously refine the remarks by Sussman above. The notion of synthesizing a "correct" program is made precise. The above excerpt from [15] is the focal point of this paper.

The research described below is also harmonious with the general plan for AI research outlined in [9]. Hofstadter believes that self reference is fundamental and basic to AI [9, pg. 714]. Self reference permeates the theory of algorithmic program synthesis presented below. The statements of the theorems involve self referential sets and the proofs (excluded from the discussion below) of many of the theorems involve writing complicated self referential programs or sequences of programs which reference themselves and other programs in the sequence.

Hofstadter's notion of filtering evidence [9, pg. 694] is also mirrored in the research described below. All the results presented below have a positive component which asserts that a programs computing functions in a particular set

can be synthesized, and a negative component which asserts that no synthesizer exists, subject to certain constraints, which can synthesize correct programs for all the functions in a particular class. The positive results are proven by exhibiting a synthesizer which, using a trivial criteria as to which evidence to ignore and which evidence to exploit, is capable of synthesizing programs for any function in a given class. Although there is nothing deep in the particular data filters that are exploited in the proofs of the results presented below, the power of using such simple filters is supportive of Hofstadter's claim that much of intelligence is deciding which evidence is valid and which is not.

This introduction has focused on how mathematically abstract research can be relevant to applied work in general. A more general (and more complete) argument for the applicability of mathematical results and proof techniques to AI is one of the major points of [9]. The next section of this paper contains the basic definitions and the general framework of a theory of automatic program synthesis. Several differentiable criteria for judging the success or failure of an attempted synthesis are introduced. The precise interrelationships between the criteria have natural interpretations with respect automatic program synthesis.

2. The General Theory

Like Sussman [15], we view automatic program synthesis as a continuing, potentially infinite, process. The data for the synthesis process is assumed to be a complete listing of the behavior of the program to be synthesized. Of course, the agent performing the synthesis can only examine finitely much of the input before producing a resultant program. Agents performing program synthesis will be called *inductive inference machines* (IIMs). Formally, IIMs are algorithmic devices which input the graph of a recursive function, an ordered pair at a time, and occasionally output programs conjectured to compute the function whose graph is being used as input [4, 8]. Hence, we consider only program

synthesizers which use an explicit description of the behavior of desired program as the input and not some more abstract problem description as envisioned in [15]. Restricting the types of valid inputs to IIMs makes the resulting theory more tractable mathematically. Elaborations on the theme using input other than the graph of a function to an IIM have been left for future research. In fact, we may furthermore suppose, without any loss of generality to the resulting theory, that the graph of a function input to an IIM is received by the IIM in the increasing domain order, i.e. $(0,f(0)), (1,(f(1)), (2,(f(2)), \cdots$. The later observation is a consequence of restricting one attention to attempting to synthesize programs for computing recursive functions.

We proceed to introduce several notions of what it means for an IIM to *successfully* synthesize a program for a recursive function f. Sussman's positive attitude about program bugs [15] and Musa's result which implies that every sufficiently complex program has at least one more bug to be found [13] suggest that we ought to consider, in some cases, the program synthesis successful even if the resultant program does not compute the input function everywhere. Rather than arbitrarily decide when a program computes a "sufficiently close" approximation to the input function we will consider various degrees of "closeness." As in [6] we say an IIM M *EX^asynthesizes* a recursive function f if M, when fed the graph of f, outputs a last program which computes f everywhere except perhaps on at most a arguments. By "a last program" we mean that at some point M outputs a program, say p, and after that point, any program output by M is syntactically identical to p, i.e. another copy of p. The "EX" abbreviates "explain" as M's final program can be construed as an explanation of the phenomenon described by f [6]. In order to compare synthesis with two errors with, say, synthesis with five errors, we collect all the sets of recursive functions which are EX^a synthesizable by some IIM into a class called EX^a. In other

words, the class EX^a is the collection of all sets **S** of recursive functions such that there is an IIM M which EX^a synthesizes programs for every function is **S**. In order to distinguish two classes, say EX^2 and EX^5, a set **S** is found to be in EX^5 and then a member of **S** is constructed for each IIM which can't be EX^2 synthesized by the chosen IIM. The first result we will discuss is from [6].

THEOREM 1.

Suppose **S** is the set of recursive functions f with the property that if f is evaluated on argument 0, the resulting output is a program to compute f everywhere except on, perhaps, at most $a+1$ inputs. Then, $S \in (EX^{a+1} - EX^a)$.

Hence, as Sussman [15] suspected, there are sets of problems (**S** in the theorem) such that no automatic program synthesizer can examine finitely much data describing any problem from the chosen set and proceed to synthesize a "correct" program, where "correct" means any desired degree of closeness (the parameter a from the theorem). Whereas Sussman [15] presents empirical evidence that there is some advantage to considering programs with bugs, and Musa [13] presents statistical evidence that it is unrealistic to abstain from using programs with anomalies, Theorem 1 proves that, in some cases, automatic program synthesizers will benefit by tolerating errors in their output.

The proof of Theorem 1 distinguishes two types of bugs in the synthesized program p with respect to the desired function f. Firstly, p may compute the wrong value on some arguments, bugs which are easily found in the limit by IIMs. On the other hand, p may *not* converge on some inputs. The later kind of bug makes it impossible for any IIM to take the program $f(0)$ and examine some finite portion of the rest of the range of f and patch up bugs. One never knows whether the program in question diverges on some input x, or is just slow in executing. Any IIM which attempt to compensate for both the defined and undefined bugs in program $f(0)$ will never output a last program, and hence

that IIM fails to EX^a synthesize all the functions in S. Note the very simple filter involved in constructing an IIM which witnesses the inclusion of S in EX^{a+1}, all information about the range of f is irrelevant except the value $f(0)$.

The notion of program synthesis described in the above does not really capture the idea of a program synthesizer which is continually revising and evolving its output based on the new inputs received. Barzdin [3] acting on a suggestion of Feldman [7] defined a notion of evolutionary program synthesis, which was independently introduced in [6]. An IIM M *BC synthesizes* a recursive function f, if when M is fed the graph of f, all but finitely many of the programs output by M compute f. The "BC" stands for "behaviorally correct." Again, we associate with BC the class of all sets of recursive functions which are BC synthesizable by some IIM. Any IIM M can be algorithmically transformed into an IIM M' which BC synthesizes any function which M can EX^a synthesize: M' simulates M on the input received and then takes the programs output by M and patches them to be correct on the finite set of inputs seen so far by M and M'; M' outputs the patched versions of the programs conjectured by M. After M' has enough data to patch all the bugs in M's last program, all the programs which M' subsequently outputs will be bug free. The next result which we will present clearly proves Sussman's claim that automatic program synthesizers which continually evolve programs are *strictly* more powerful than synthesizers which complete their tasks and then ignore any later arriving data.

THEOREM 2.

Suppose S is the set of recursive functions f which, on all but finitely many inputs, output programs which compute f. Then every function in S can be BC synthesized by a single IIM. Furthermore, no IIM can EX^a synthesize all the functions in S, for any natural number a.

A special case of Theorem 2 appears in [3] and a generalization, obtained in collaboration with L. Harrington, appears in [6]. In fact, a much stronger results holds. To state the stronger result we need to define what it means for a finite collection of IIMs to synthesize a set of functions. As in [14] we say IIMs $M_0, M_1, \cdots, M_n$ *EX^asynthesize* all the recursive functions in some set S if for each $f \in S$ there is an $i \leq n$ such that M_i EX^a synthesizes f. Furthermore, S is a member of the class $C(n+1, EX^a)$ along with any other set of recursive functions which can be EX^a synthesized by a collection of at most $n+1$ IIMs. The following theorem is from [14].

THEOREM 3.

Let **S** be as in the previous theorem. Then **S** is **not** a member of $C(n+1, EX^a)$ for any natural numbers n and a.

The above result may be slightly misleading as the classes BC and $C(n+1, EX^a)$ are incomparable [14]. That is to say there are sets of recursive functions for which programs can be synthesized by collections of IIMs which do not evolve programs but cannot be synthesized by an IIM which does evolve programs continually over time. Hence, there is a potential advantage to be gained by using evolutionary program synthesizers and a different sort of advantage which can be obtained by using multiple synthesizers. Furthermore, the different types of advantages alluded to in the previous sentence are incomparable.

The issue of multiple machine program synthesis deserves further discussion. Returning to synthesized programs with (some small number of) bugs, one would expect that any set of recursive functions which can be synthesized by collection of, say, three IIMs with at most five bugs in the resulting programs could be synthesized more accurately (at most two bugs) by some larger collection of IIMs. Not only can more accurate programs be synthesized by using more IIMs, it can be determined how many more IIMs are required to achieve a

desired improvement in accuracy, as shown by the following result from [14].

THEOREM 4.

The class $C(m,EX^a)$ is included in the class $C(n,EX^b)$ if and only if $n \geq m \cdot (1 + \lfloor a \div (b+1) \rfloor)$.

As mentioned above, there are positive and negative components to results such as the above theorem. To prove part of Theorem 4 it is necessary to show that some set of recursive functions, **S**, can be synthesized by some m-tuple of IIMs but not by some other n-tuple of IIMs. To show that all the functions in **S** can be synthesized by by some m-tuple of IIMs, two trivial data filters are used. Firstly, each function in **S** is divided into m disjoint pieces with each IIM examining exactly one such piece, with no piece being considered by more than one IIM. Another filter is used by each of the m IIMs in its examination of its piece of the input. Again, we find more evidence supporting Hofstadter's views on the importance of data filters [9]. Other results in [14] illustrate a critical mass principle for automatic program synthesis. That is, there are sets of functions which can be synthesized by some $n+1$ tuple of IIMs but which cannot be synthesized by any n tuple of IIMs. So, for machines which learn by example, diversity of perspective may be crucial for some tasks.

In our initial discussion of the synthesis of programs with bugs above it was noted that, in general, one can't patch errors in a program and know when to stop patching even if there was an apriori bound on the number of bugs in the original program. The final result which we discuss in this paper indicates that one can not, in general, take an arbitrary program synthesizer, look at an proposed solution for some input, give the synthesizer some data and expect the synthesizer to subsequently output a more accurate solution. As before, we start by defining some more criteria for successful synthesis of programs by IIMs. The mathematical distinction of the criteria yields the desired

interpretation. As in [6] we say an IIM M EX_b^a *synthesizes* a recursive function f if M EX^a synthesizes f after outputing at most $b+1$ distinct programs. Note that for EX_b synthesis it is required that the IIM change its conjecture as to a solution at most b times. The class EX_b^a contains all the sets of recursive functions which are EX_b^a synthesizable by some IIM. The promised interpretation is derived from the following theorem from [6].

THEOREM 5.

The class EX_b^a is included in the class EX_d^c iff $a \leq c$ and $b \leq d$, for any natural numbers a, b, c, d.

In fact the EX_b^a classes form a lattice under strict inclusion. We conclude this section with a discussion of some generalizations the synthesis criteria discussed above. In the definition of EX_b^a above a and b were assumed to be natural numbers. One extension is to allow a and or b to be "*" with the meaning "finitely many". For example, EX^a synthesis is synonymous with EX_*^a synthesis. If M EX^* synthesizes f then M on input from the graph of f outputs a last program which computes a finite variant of f. The above theorems all hold if * is interpreted like an integer which is larger than any natural number. The BC synthesis of programs with bugs has also been investigated [6]. BC^a denotes the class of all sets S of recursive functions for which there is an IIM M such that if M is fed the graph of any f in S, all but finitely many of the programs output by M compute f, except on at most a inputs. Different programs output by M may differ from f on different inputs. A hierarchy unfolds by allowing more and more bugs. L. Harrington has constructed an IIM which BC^* synthesizes every recursive function (see [6]).

3. Cognitive Styles of Learning

This section is devoted to comments on various cognitive styles of learning by example. Even though the comments are based on the results mentioned in the previous section, the remarks below are more speculation than science. In order to draw conclusions as to the nature of how people learn by example from our theorems, one must assume that the brain is a complicated machine. Although this perspective is unpopular with a majority of the population, adopting a mechanistic viewpoint is a prerequisite for engaging in artificial intelligence research.

Consider a successful $EX^0_?$ synthesis performed by an IIM M. M conjectures some finite number of programs before converging on one which computes the input function everywhere. M conforms to the standard method of empirical science, i.e. gather more data, test the current conjecture against the new data and issue a new conjecture, gather more data ... until a conjecture is made which is never later refuted. M's behavior exemplifies iteratively refining a concept until perfection is achieved, a commonly observed cognitive style in humans. Next consider M successfully performing an EX^*_0 synthesis. M examines some data and then in a "flash of inspiration" outputs an answer which is close to the desired result. It was Holmes [10] who first said "A moment's insight is sometimes worth a life's experience." His intuition is analogous with the consequence of our theorems which indicate that there is a set in the class EX^*_0 which is not in the class $EX^0_?$. In this case, M's cognitive style can be classified as that of an intuitive thinker who pays scant attention to detail.

The classes EX^*_0 and $EX^0_?$ are extreme points in the EX lattice. The other points in the lattice represent cognitive styles which are a mixture of the iterative perfectionist and the intuitive approximator. As the EX lattice result indicates, any mixture is possible and results is unique learning capabilities. How-

ever, I suspect that people learn many things in a behaviorally correct fashion. That is, people tend to change their minds frequently, even when they are not confronted with evidence which directly contraindicates their previous thoughts. Furthermore, it is only human to resist changing ones beliefs when confronted with an instance for which they are inadequate or incorrect.

4. Conclusions

We have interpreted inductive inference theorems for automatic program synthesis. The resulting interpretations were consistent with, even refinements of, other ideas which have appeared in the AI literature. The theorems mentioned above are a scant sample of results in inductive inference which are applicable to the more practical side of automatic program synthesis. A more ambitious exposition of the application of inductive inference to automatic program synthesis is planned [2]. In summary, most of this paper has been devoted to drawing conclusions about automatic program synthesis from theoretical results in the in the more general area of inductive inference.

References

1. AHO, A., *Currents in the Thoery of Computing*, Prentice Hall, Englewood Cliffs, N.J. (1973).
2. ANGLUIN, D. AND SMITH, C. H., *A discrete theory of automatic program synthesis*, In preparation 1981.
3. BARZDIN, J., "Two theorems on the limiting synthesis of functions," pp. 82-88 in *Theory of Algorithms and Programs*, ed. Barzdin,Latvian State University, Riga, U.S.S.R. (1974). 210
4. BLUM, L. AND BLUM, M., "Toward a mathematical theory of inductive inference," *Information and Control* **28** pp. 125-155 (1975).
5. BLUM, M., "A machine-independent theory of the complexity of recursive functions," *JACM* 14(1967).
6. CASE, J. AND SMITH, C., *Comparison of identification criteria for machine inductive inference*, submitted for publication. 1981.
7. FELDMAN, J., "Some decidability results on grammatical inference of best programs," *Math Systems Theory* **10** pp. 244-262 (1972).
8. GOLD, E.M., "Language identification in the limit," *Information and Control* **10** pp. 447-474 (1967).
9. HOFSTADTER, D. R., *Gödel, Escher, Bach: an eternal golden braid*, Basic Books, New York (1979).
10. HOLMES, O. W., *The Professor at the Breakfast Table*, Houghton Mifflin Co., Boston, Mass. (1982).
11. MACHTEY, M. AND YOUNG, P., *An Introduction to the General Theory of Algorithms*, North-Holland, New York, New York (1978).
12. MINSKY, M., *Computation: Finite and Infinite Machines*, Prentice Hall, Englwood Cliffs, N.J. (1967).
13. MUSA, J. D., "A theory of software reliability and its application," *IEEE Transactions on Software Engineering* **SE-1**(3) pp. 312-327 (1975).
14. SMITH, C. H., *The power of parallelism for automatic program systhesis*, Submitted for publication. 1981.
15. SUSSMAN, G. J., *A Computer Model of Skill Acquisition*, American Elsevier Co., New York (1975).
16. TURING, A., "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society* **42,43** pp. 230-265,544-546 (1936).